BUILDING DATA STORAGE AND ANALYTIC BACKEND SERVICES FOR LISTEN ONLINE

BY

ZHAOHENG HU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Kevin Chen-Chuan Chang

ABSTRACT

Data storage and post-processing are among the most important data-related tasks. These tasks aim at keeping the data available and reusable in the long term so people can look into the data, manipulate the data and find valuable information that they need. The tasks can be more complex and difficult to deal with when the domain of the problem expands to the Big-Social World. In this case, the data could be nonuniform, which means the source of data is not limited to only one social media and the structure of the data could be variant. Therefore, traditional relational database management systems (RDBMSs) cannot properly work here to handle the unstructured data.

This thesis introduces a system which integrates both Neo4j, a graph database, and MySQL, a traditional relational database, together to solve the unstructured social media data management problem mentioned above. The system has been integrated in Listen Online, or Lion for short, to handle the real problems. For convenience, we call the system Lion-Backend. Lastly, by building upon some existing libraries, Lion-Backend provides graphical interfaces to users to help them easily build their queries and apply analysis functions to their data.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I have been on this attractive project during my whole graduate study and there are innumerable people who helped me and gave me confidence, ideas and solutions.

The fist person I would like to acknowledge is my adviser, Professor Kevin Chang. He conceived this "Social Universe" project initially and met with our project team weekly to listen to our progress and new ideas. He tried to introduce and present our work to others and collect feedback for us so we could improve our project.

In addition, I acknowledge all my hardworking team members: Dustyn James Tubbs, Chen Wang, Lunan Li, Zongyi Wang and Naijing Zhang. All of them made their own contribution so we could work as a team to finish this project.

Lastly, I acknowledge the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. I came here when I was an undergraduate student. During the past years, I acquired knowledge and made new friends here. I can never forget such a meaningful and valuable time.

TABLE OF CONTENTS

1. INTRODUCTION

This thesis introduces a system (Lion-Backend) that solves the data management issue in Big-Social World. Before talking about the details of Lion-Backend, we need to better understand the background and the problem we are facing.

1.1 Big-Social World

With the power of the Internet, social media is becoming one of the essential elements in human daily life. People spend lots of time on social media every day. For example, Facebook has more than 400 million active users and about 50% of them log on to Facebook on any given day. People also spend more than 55 minutes on Facebook daily [1]. The image below shows the percentages of Facebook users who use Facebook daily, weekly or monthly (Fig 1.1). Therefore, people can generate a huge volume of data each day since all their activities like following, liking and posting are recorded. Given all this, we can say that there is a world consisting of big data and human information is included in this world implicitly or explicitly. We call this world Big-Social World.

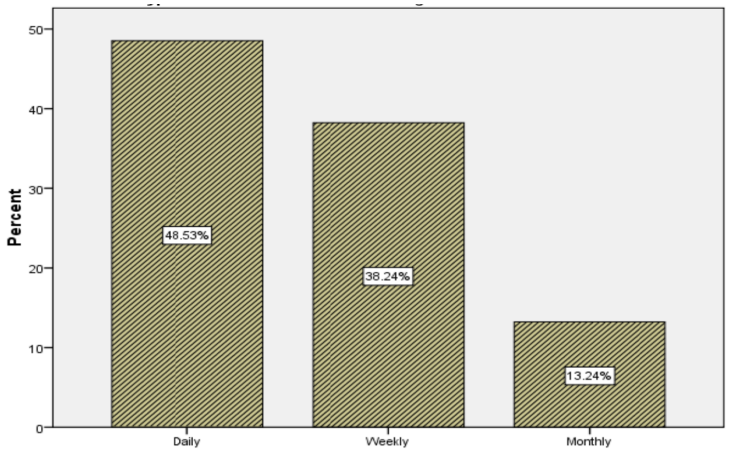

Figure 1.1: Frequency of Facebook use

1.2 Database System

A database management system (or DBMS) is a computerized data-keeping system [2] that makes it possible for end users to create, read, update and delete data in a database. A DBMS is necessary in the big data management and processing problem since it is not scalable to request all the data from data resources each time, and a DBMS can make the data accessible and usable repeatedly and efficiently.

Databases can be classified in two different types: relational database and non-relational database. A typical relational database like MySQL, PostgreSQL or SQLite3 represents and stores data in tables and rows while non-relational databases like MongoDB present data in some other structures including JSON, graphs and so on.

Since the scope of our problem is in the social universe where entities like person, posting, location, etc., are connected through different directed or undirected relationships and the mapping between entities could be one-to-one, one-to-many or many-to-one, it is normal to think about the Big-Social World as a huge graph in which nodes are representing the entities and edges are the relationships between them (Fig 1.2).

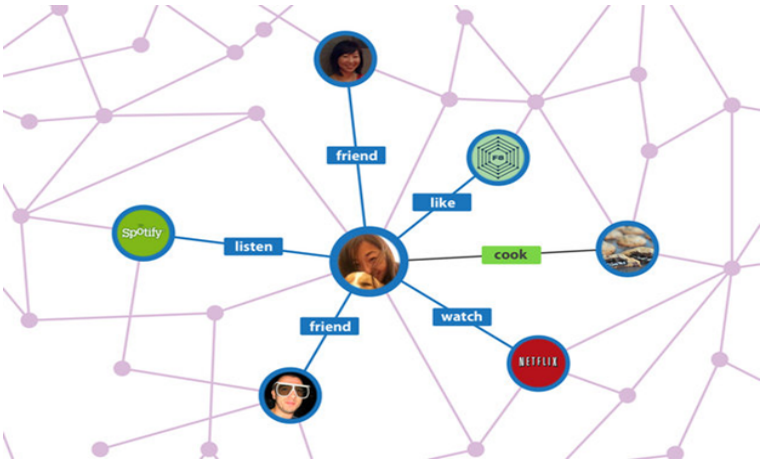

Figure 1.2: Big-Social World as a graph (Dickinson, 2012 [3])

Due to such facts and some key rules for digital data storage [4], it is necessary to keep the raw data raw and structured for analysis. Therefore, Lion-Backend is built based on Neo4j [5], a non-relational database where data are organized in graph structure. In Neo4j, everything is stored as either an edge, a node or an attribute where nodes and edges can be labeled to narrow down the searching scope. There are several reasons to use Neo4j. On the one hand, from the description above we can see that the high similarity between Neo4j data structure and Big-Social World entity structure makes Neo4j an appropriate container of data. On the other hand, the query language of Neo4j, which is called Cypher, is human readable and easy to learn since a Cypher query can intuitively describe what kind of data is being looked for. An example of Cypher query is as follows:

MATCH (p:Person)-[r:livesIn]→(c:City {name:'Chicago'}) RETURN p;

The above Cypher query searches for people who live in Chicago. The example shows that the structure of Cypher language is similar to relationships in the real world, which makes it easy to understand.

1.3 Data Pre-processing

Data from different social media resources in Big-Social World could be variant. Therefore, data need to be pre-processed properly to fit into the Neo4j database. Since the data provided by resources are all organized in

JSON structure, Lion-Backend makes use of open library Genson to uniformly parse the schema of JSON, convert the data to nodes and build edges between them to represent the relationship.

A toy example of such conversion looks like Fig 1.3. Data pre-processing refers to the 2^{nd} step which builds a bridge that transforms JSON data to nodes and edges in the Neo4j database so they can be used for further purposes.

1.4 Data Post-processing

The final target of the Lion-Backend is to help users including researchers and data scientists to extract valuable information that can be used in various areas. Therefore, post-processing on data like computation and visualization is one of the key steps in the workflow. In Lion-Backend, several post-processing tools are designed to help users overcome the difficulties that they may meet due to technical barriers. Users will be able to process and analyze their data in multiple ways including textual analysis, numerical computation, etc. These uses will be illustrated in more detail later.

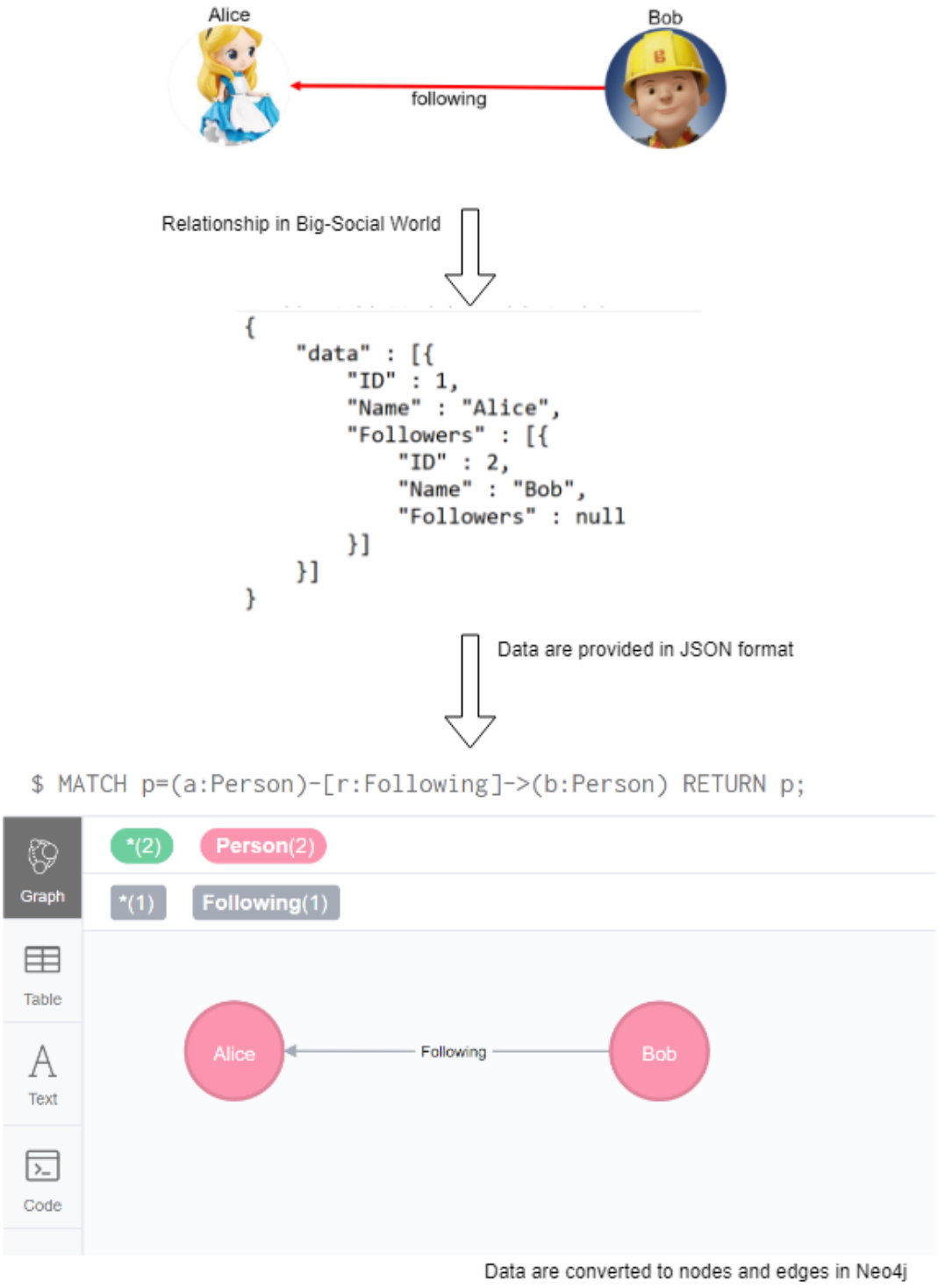

Figure 1.3: Transformation from real relationship in Big-Social World to nodes and edges in Neo4j

2. RELATED WORK

During the building of Lion-Backend, some related projects were found. The purposes or use cases of these projects may not be the same as ours, but the ideas and technologies are valuable and inspirational to developers. In this chapter, we will briefly summarize the related work..

2.1 Genson

The first thing after catching the data from sources is storing them for long term use. The point here is how to translate the data to the structure that can fit into Neo4j database. However, due to the fact that the number of social media sources could be large and the content of JSON data could be totally different, implementing a translator for each of them could be time-consuming and error-prone. Therefore, in order to implement one structure that works for all, the JSON data should be parsed dynamically instead of using pre-defined schema. In this case, Genson is a helpful tool which is integrated in Lion-Backend.

Genson is a powerful JSON schema generator built in Python [6]. Given data in JSON format, what Genson will do is not directly process the data and convert to a specific data structure, but rather provide an answer as to what this JSON data looks like. For example, Genson could provide information about the data such as the type of each value, the name of each key, whether the key is required or nullable, etc. Fig 2.1 and Fig 2.2 are simple examples of using Genson to understand JSON. Fig 2.1 displays the original JSON data and the structure of data described by Genson is shown in Fig 2.2.

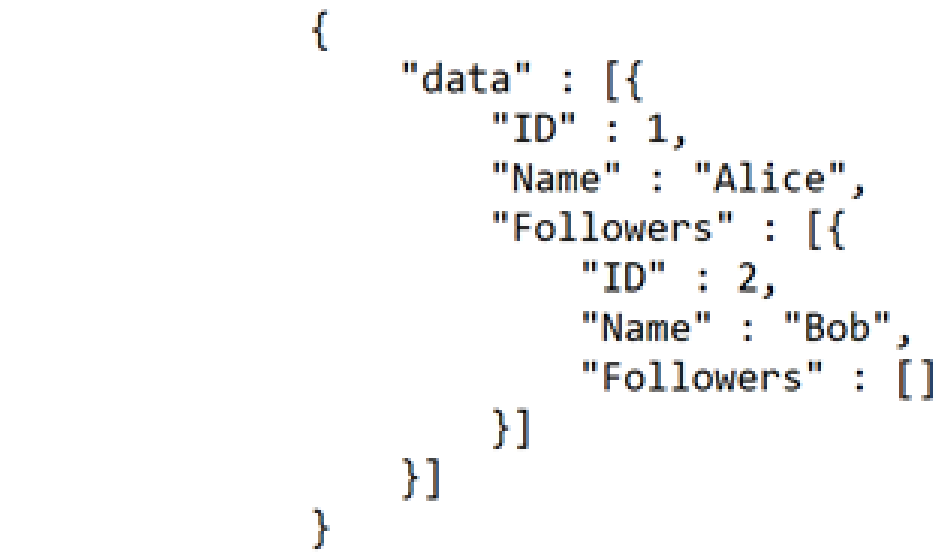

Figure 2.1: Simple example of original data

Figure 2.2: Structure of data described by Genson

2.2 Popoto.js

Lion-Backend aims to help all users to easily access the social media data they collected and process them. Therefore, it is important to help them query their data from the Neo4j database even if they have no database-relevant background. Here the idea is to provide users with a graphical interface which is intuitive and easy to understand and operate. With such an interface, simply clicking mouse, pressing a keyboard and setting up some configurations could be enough for users to build a query, instead of writing the whole query by themselves.

Popoto.js is a JavaScript library built with D3.js designed to create an interactive and customizable visual query builder for Neo4jgraph databases. The graph queries are translated into Cypher and run on the database. Popoto also helps to display and customize the results [7]. However, Popoto.js still has some limitations like insufficient data processing tools, hardcoded starting point, etc. Fig 2.3 shows an example of query builder, and the corresponding result is displayed in Fig 2.4.

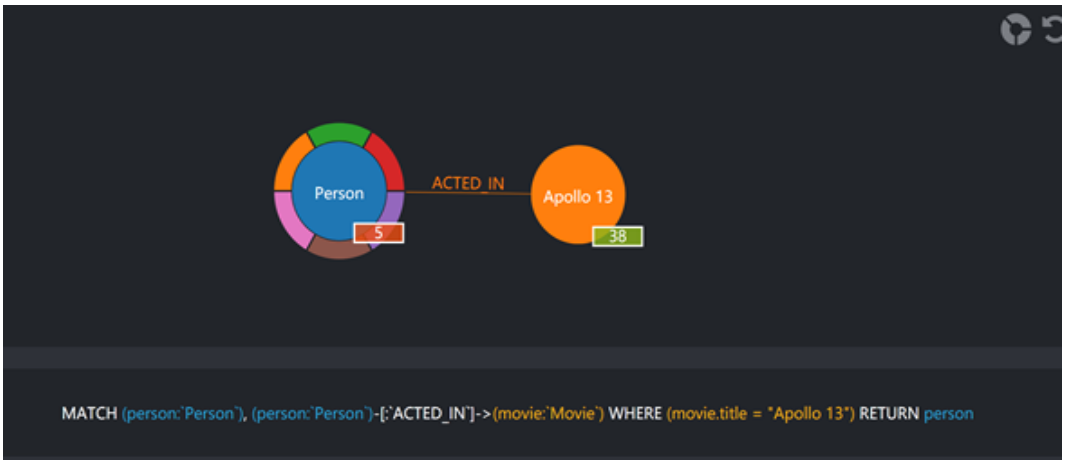

Figure 2.3: Example of Popoto.js query builder

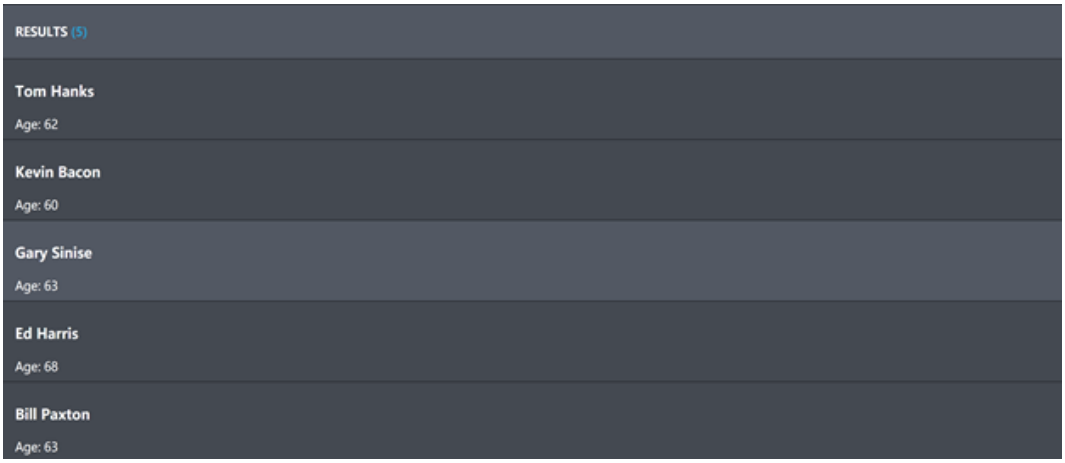

Figure 2.4: Result of example query

It could be found that the result is not that useful since a researcher who is interested in Big-Social World needs to do many operations like filtering, rather than just query the data and look at them. As a consequence, Lion-Backend is not built based on Popoto.js. In order to support data processing work better, we built our own customized interface and backend logic. The details will be explained in Chapter 3.

3. DESIGN AND IMPLEMENTATION

As stated previously, users of Lion-Backend have the following characteristics:

- Prefer to make use of data that comes from multiple social media sources with variant structure.
- Lack knowledge of programming, so they may feel hesitant to query data from the database directly.
- Are interested in Big-Social World and want to find useful information by processing the data as they want.

Based on these characteristics, Lion-Backend is designed to have the architecture shown in Fig 3.1.

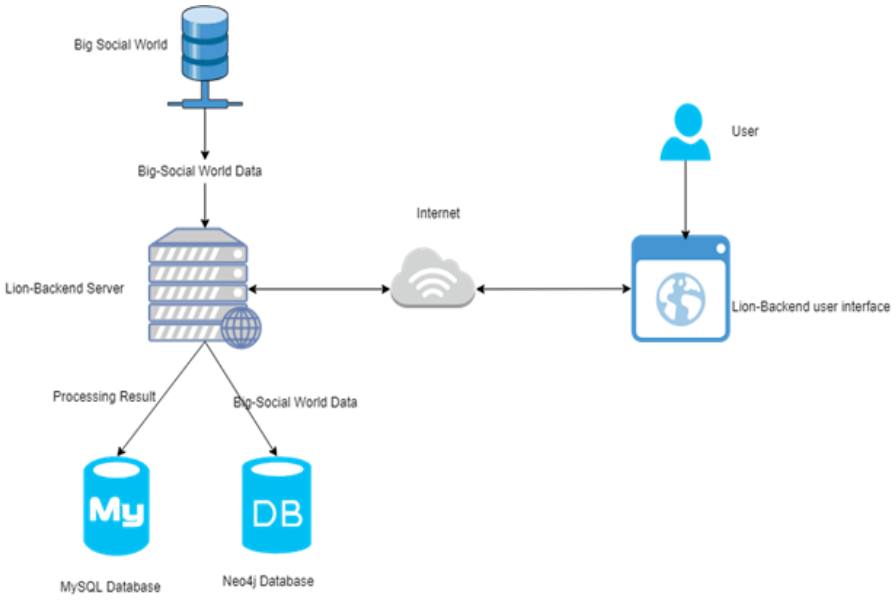

Figure 3.1: Architecture of Lion-Backend

Also, Fig 3.2 and Fig 3.3 show the details of server and user interface in Lion-Backend.

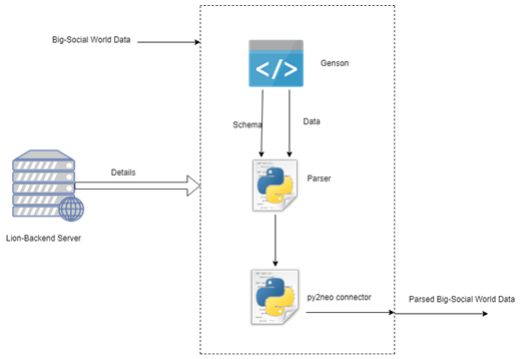

Figure 3.2: Inner structure of Lion-Backend server

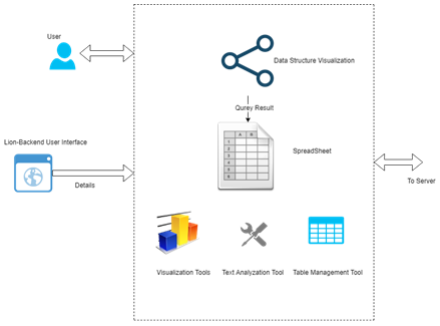

Figure 3.3: Inner structure of Lion-Backend user interface

In this chapter, we will describe the whole system by introducing key components one by one to better understand the functionalities and workflow of Lion-Backend.

3.1 Unstructured JSON Data Parser

Although structures of data from different resources could be different, all of them could be described by Genson as we mentioned above. (For convenience, we call the output of Genson that describes the data structure "descriptor" in the remainder of the chapter.) Therefore, in order to uniformly process the data and load to the Neo4j database, we perform the following algorithm which takes both data and descriptor as input to pre-process the data and send them to Neo4j properly.

Algorithm 1 An algorithm to pre-process and store data based on descriptor

```
1: procedure storeData(data, descriptor, nodeName, parendID)
2:     if data = None then
3:         return
4:     else
5:         dataType=descriptor[type]
6:         if dataType is a list or dataType='array' then
7:             dataType = first non-null value
8:             storeData(data, dataType, nodeName, parendID)
9:         else if dataType = 'object' then
10:            query = buildQuery(parentID, nodeName)
11:            newID = executeQuery(query)
12:            properties = descriptor['properties']
13:            for newNode, newDescriptor in properties do
14:                storeData(data[newNode], newDescriptor, newNode,
    newID)
15:            end for
16:        else
17:            query = buildQuery(parentID, nodeName)
18:            newID = executeQuery(query)
19:        end if
20:        return
21:    end if
22: end procedure
```

In algorithm 1, the descriptor can generate a *dataType* variable and the data will be handled differently based on the *dataType*, i.e., whether or not it is primitive:

- Primitive data ($dataType = int$, $float$, $string$, ...) is stored as a $Value$ node where there is a property called "value" to store the exact value.
- Non-primitive data ($dataType = array$, $object$, ...) is stored as an $Object$ node where there is a $nodeName$ property to store its name, and its value can be found by following the $hasChild$ relationship to find the final $Value$ node.

The *storeData* is a function which is called recursively to convert the object to node one by one. During each *storeData* function call, once an object is found and its properties are handled, *buildQuery* function will be called to generate the corresponding Cypher query, and then *executeQuery* can execute this query to store it as a node in Neo4j. *ExecuteQuery* will return the unique ID of the node generated by Neo4j after the object is stored, and this ID will be passed to the next *storeData* function call. Therefore, *buildQuery* can take this ID as $parentID$ to connect this node to its children. Another argument taken by *buildQuery* is $nodeName$, which represents the name of the current node to be stored, and it can be found by accessing the descriptor in current *storeData* function call.

To be more intuitive, Fig 3.4 and Fig 3.5 illustrate an example. The JSON data displayed in Fig 3.4 will be finally stored in Neo4j and the result can be found in 3.5.

```
{
    "Person" : [{
        "Name" : "Alice"
    }, {
        "Name" : "Bob"
    }]
}
```

Figure 3.4: Example JSON data

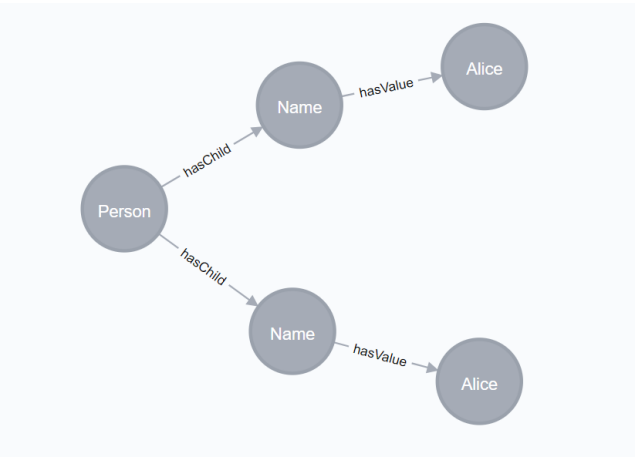

Figure 3.5: Result of pre-processed data stored in Neo4j

3.2 Visual Query Builder

Once data are successfully parsed and stored into Neo4j database on the server side, another key point is to help users easily access their data. Visual query builder is the component that works for this. Visual query builder can show users the structure of data in Neo4j and interact with users to receive their inputs. After that, users' inputs can be parsed and the queries will be built correspondingly. The visual query builder consists of the following parts.

3.2.1 Graphical User Interface (GUI)

In visual query builder, the GUI is the place where the user can interact to visually learn about the data structure and build their query. An example screenshot is in Fig 3.6.

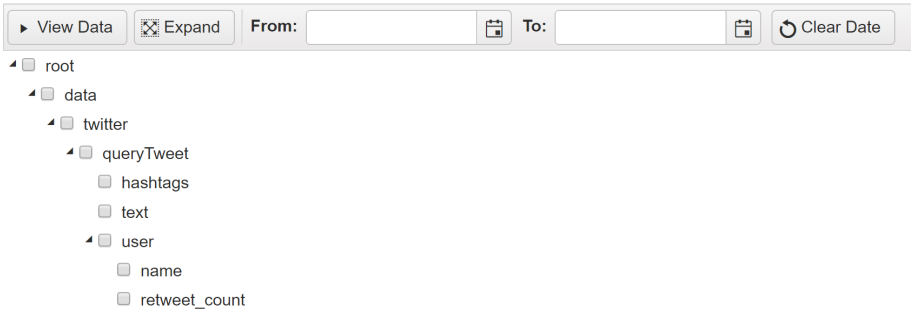

Figure 3.6: Example GUI in visual query builder

From the previous section, algorithm 1 will parse the nested JSON data into parent-children relations, which means data are organized in a tree-like structure. Thus, the GUI uses a tree-view component to visualize the data structure to users. Fig 3.6 shows that with such a displaying approach, it is intuitive to figure out the layout of each query. Also, checkboxes are added to the GUI so users can construct their queries by simply checking the elements they want to query about in the tree-view.

3.2.2 Tree-View to Queries Converter

After users selected the elements they want to query about, the selections will be passed from user interface to Lion-Backend server. In this case, the next step is parsing the selections on the server side, building the real Cypher queries based on them and executing the queries. In Lion-Backend, we apply several algorithms for different purposes and combine them together to finally achieve the goal. Algorithm 2 is to find all paths that end up with a checked checkbox in the tree-view. Algorithm 3 is to convert all paths found in algorithm 2 to Cypher queries; this algorithm will call algorithm 4 as a helper.

Algorithm 2 An algorithm to find paths that end up with a checked checkbox

```
1:  procedure findCheckedPaths(checkedCheckboxes)
2:      ret = []
3:      for checkedCheckbox in checkedCheckboxes do
4:          path = [checkedCheckbox]
5:          parent = checkedCheckbox.parent
6:          while parent ≠ undefined do
7:              path = [parent] + path
8:              parent = parent.parent
9:          end while
10:         ret.append(path)
11:     end for
12:     return ret
13: end procedure
```

Algorithm 3 An algorithm to convert paths to Cypher queries

```
1: procedure queryBuilder(paths)
2:     for path in paths do
3:         queryBuilderHelper(path, rootID)
4:     end for
5: end procedure
```

Algorithm 4 Helper for queryBuilder

```
1: procedure queryBuilderHelper(path, rootID)
2:     query = prefix
3:     for each in path do
4:         query += -[:hasChild]→(:Object{node_name: each})
5:         if isLastOne then
6:             query += -[:hasValue]→(v:Value) RETURN (v)
7:         end if
8:     end for
9:     return query
10: end procedure
```

In algorithm 4, the initial value of the query will be a fixed prefix: *MATCH (r:Root) WHERE ID(r) = rootID WITH (a) MATCH (a)*. With this initial value, a single path will be iterated node by node and the query will be appended during accessing each node. Finally, at the last step, a $Value$ node will be added as the target of this query. In this way, a path will be traversed until the last $Value$ node to completely construct the corresponding query, and algorithm 3 will call this helper multiple times to build all queries.

3.2.3 Data Displaying

Once all queries are generated, there is a connector called *py2neo* to connect the server with the Neo4j database, execute these queries and collect the result. *Py2neo* is a client library and toolkit for working with Neo4j from within Python applications and from the command line [8]. In the end, the resulting data will be passed back to the Lion-Backend user interface mentioned previously.

In order to display the resulting data, the Lion-Backend user interface makes use of a spreadsheet which is a web tool to show and manage data just like Excel. The reason that spreadsheet is chosen is that almost everyone is familiar with Excel so there should not be too much difficulty learning how to use the Lion-Backend user interface, which can improve the user experience. Also, some researchers think that using a spreadsheet can improve their productivity [9].

An example of displaying resulting data in the user interface can be found in Fig 3.7.

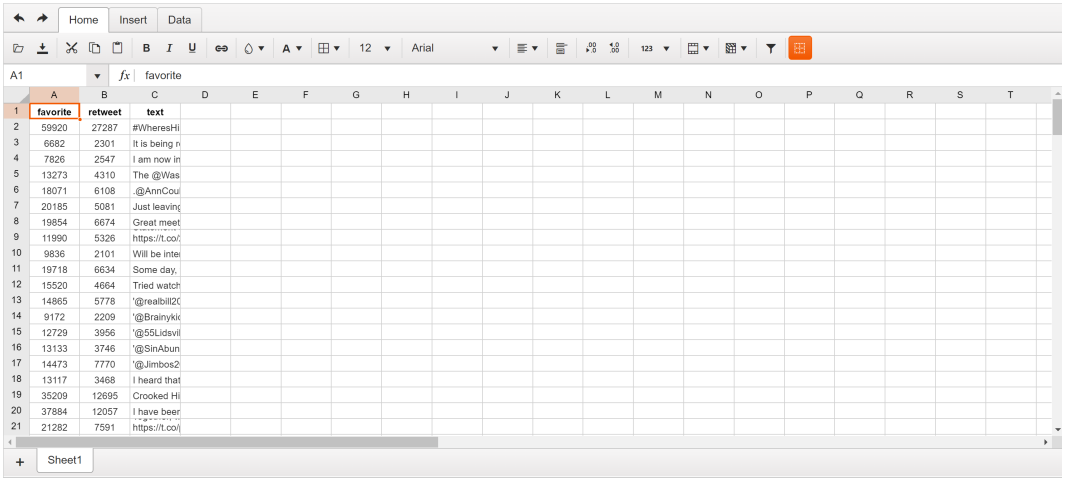

Figure 3.7: Example of using spreadsheet to show query result

Additionally, in Lion-Backend, many advanced functions are supported based on the spreadsheet and they will be introduced later.

3.3 Visualization Tools

The visual query builder component can help users to easily build and execute the query they need, and the results of the queries are displayed in the spreadsheet. However, these are not useful enough to researchers or others who are interested in data to understand the data.

Therefore, some plotting tools for visualization are provided in Lion-Backend. The tools are integrated with the spreadsheet, so simply selecting the data and pressing the visualize button are enough. Currently the plotting tools can plot the data in bar chart, pie chart, line chart and column chart. More features could be considered in future work.

Fig 3.8 shows an example of using the column chart tool to visualize the retweet counts of some of Trump's tweets.

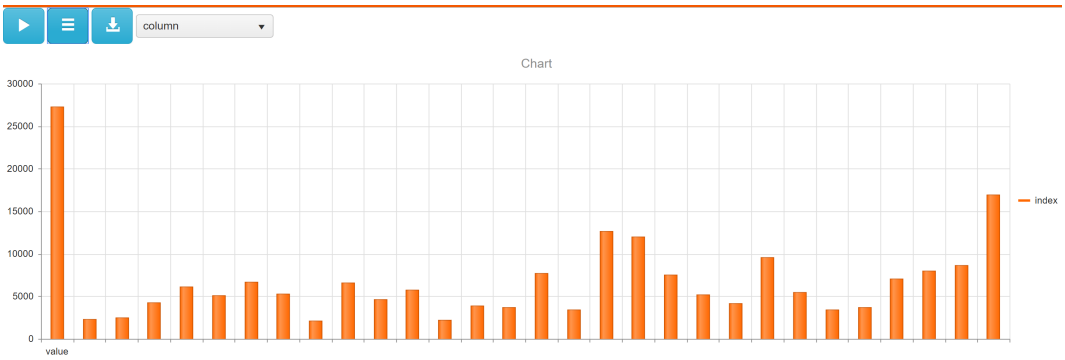

Figure 3.8: Example of visualization

3.4 Text Analysis Tool

The visualization tools introduced in the previous section are used for numerical data. However, a large portion of data in Big-Social World is non-numerical data like text, images, etc. In such cases, Lion-Backend also has some tools which could be used in text analysis. Some of these tools are built based on Aylien Text Analysis API [10], which can be helpful in natural language processing for effective understanding of human-generated text.

For now, the provided tools include figuring out sentiment, extracting concepts, entities, hashtags and computing top-K words. Fig 3.9 shows the result of extracting hashtags from some of Trump's tweets.

Figure 3.9: Example of hashtag extraction

3.5 Customized Formula

A key point of providing a good user experience is reducing redundant work that needs to be done by users. Redundant work can happen when a user wants to repeat a certain computational operation which is not directly provided in spreadsheet formulas every time. In order to avoid this, Lion-Backend provides an interface to those users who have programming skills to write code and create their own formula.

In the interface, the user can name the formula he/she is creating and click "add argument" button to add an argument and define its type for this formula. After all arguments are added, by clicking "assign arguments", the above configurations will be applied and the signature of the formula will be generated in the coding area so the user can write the code to implement the detail. An example can be found in Fig 3.10. In this example, we declared a new formula called $absOfDiff$ which has two arguments, $cell1$ and $cell2$. The implementation is done in the coding area. Just as its name suggests, this formula takes two cells as inputs, gets their values and computes the absolute value of the difference between them. Once all these steps are done, the user clicks "submit" to effectuate the code.

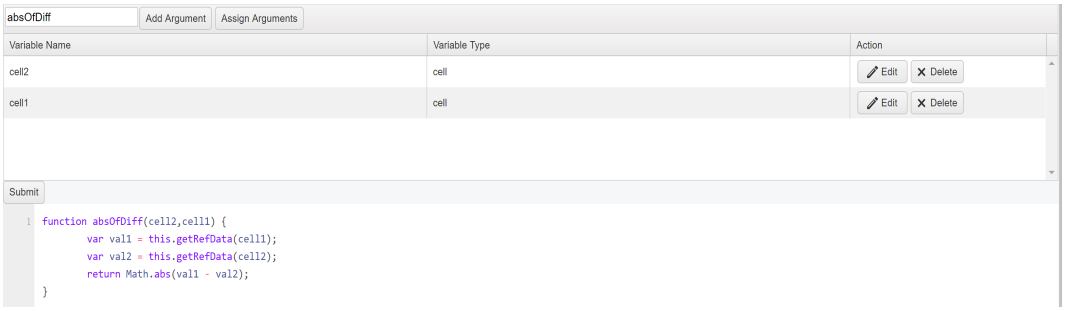

Figure 3.10: Example of defining formula

The result of submitting the $absOfDiff$ formula and using it in the spreadsheet can be seen in Fig 3.11.

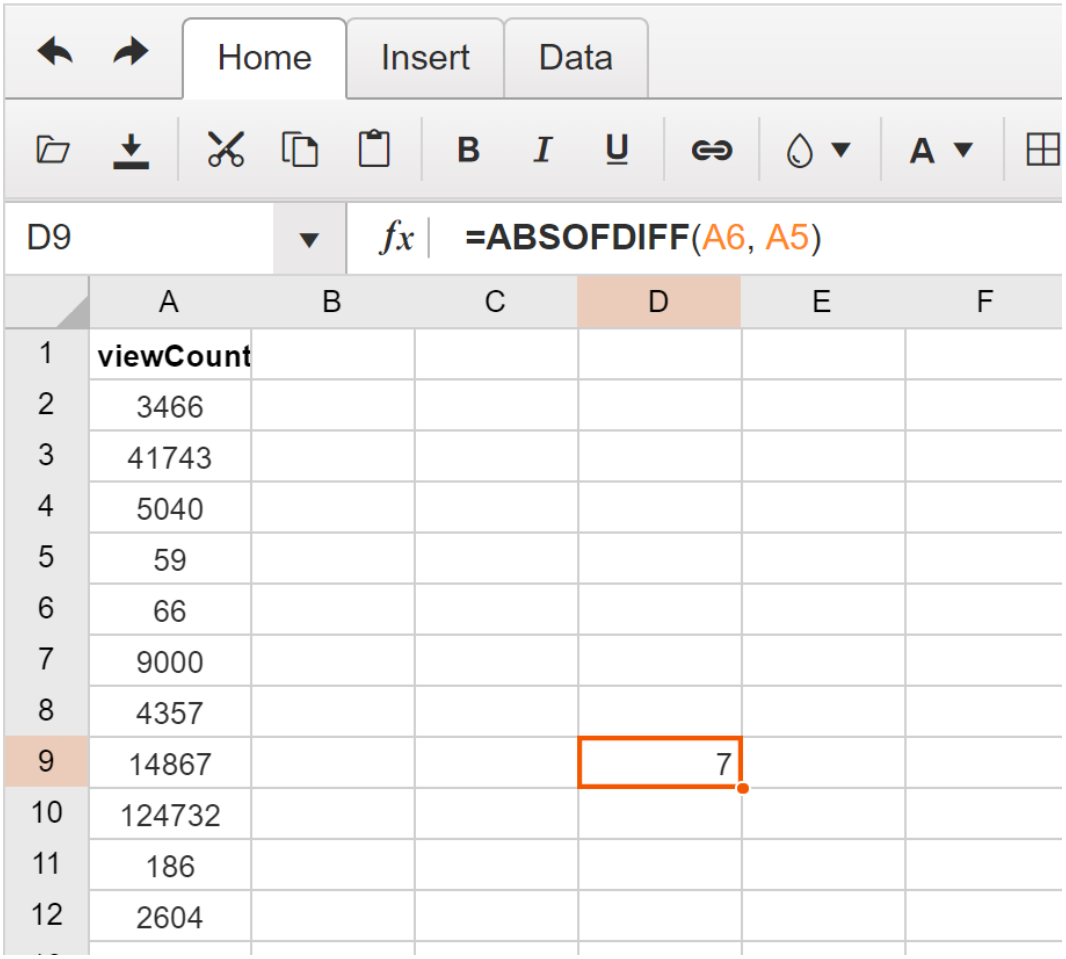

Figure 3.11: Result of absOfDiff

3.6 Connect with MySQL

Big-Social World consists of connections between entities where an entity could be a person, a post, a place, etc. A researcher who is using Lion-Backend may need to find connections between entities. For example, a researcher may want to find the YouTube videos and tweets which are connected by a common topic. In such a case, he/she will try some operations including writing a program to get a list of YouTube video topics and iterating tweets to see which tweet contains one of those topics in its text. This could be hard and boring to those who know little about programming.

However, these problems can be solved in Lion-Backend because the spreadsheet is directly connected with a MySQL database.

3.6.1 Why MySQL

One can have an intuition about the example described above that the operation there is a join operation. Therefore, importing the data from spreadsheet to a MySQL database will make the task much easier since instead of writing a program, a simple query like

*SELECT * FROM tweets JOIN videos ON tweets.text LIKE CONCAT('%', videos.topic, '%');*

can solve the problem efficiently. Additionally, while processing data with the spreadsheet, it is important to make regular backups of data [11]. Therefore, in Lion-Backend, MySQL is not only a tool to process complex operations by allowing users to write MySQL queries, but also a container to safely store the data and processing result permanently for future reuse.

3.6.2 How to Import Data

A user may not feel comfortable using MySQL by themselves due to the fact that users need to design schema, create tables, and set columns for each table they want to process, which means it is redundant, boring and error-prone. In Lion-Backend, algorithm 5 is applied to the data collected from the spreadsheet so data can be converted to proper format for the server side to process and store to MySQL.

In algorithm 5, the return value, called $parsedSheet$, is an array of objects. Each object in $parsedSheet$ consists of several key-value pairs where key is the name of the column and the value is the corresponding value. Here the value is converted to string type to uniformly handle the variant types of data. Thus, the data will be stored as $TEXT$ type in MySQL database.

After parsing, $parsedSheet$ is passed to the server together with the column list, username and user-defined table name (called $originalTableName$ below). Lion-Backend server will correspondingly create the table with name $derivedTableName$ where $derivedTableName$ is the result of concatenating $username$, _(underscore) and $originalTableName$. Lion-Backend uses $derivedTableName$ instead of $originalTableName$ to distinguish those tables with the same name but created by different users.

All operations and algorithms are done by the Lion-Backend system. For users, the only thing they need to do is click "Save Current Sheet" so the amount of the work that needs to be done is much less.

Algorithm 5 Helper for queryBuilder

```
1:  procedure parseSheet(data, cols)
2:      parsedSheet = []
3:      for row in data do
4:          parsedRow = {}
5:          for index, column in cols do
6:              value = row[index]
7:              if value ≠ null then
8:                  parsedRow[column] = String(value)
9:              else
10:                 parsedRow[column] = null
11:             end if
12:         end for
13:         parsedSheet.append(parsedRow)
14:     end for
15:     return parsedSheet
16: end procedure
```

3.6.3 How to Perform Complex Operations

After the connection between the spreadsheet and MySQL has been set up, and the data has been successfully and efficiently stored, Lion-Backend provides a coding interface for those who want to perform complex processing operations on data so they can write their own queries as they want, as long as they are supported by MySQL. Fig 3.12 is an example of executing a MySQL query on stored table.

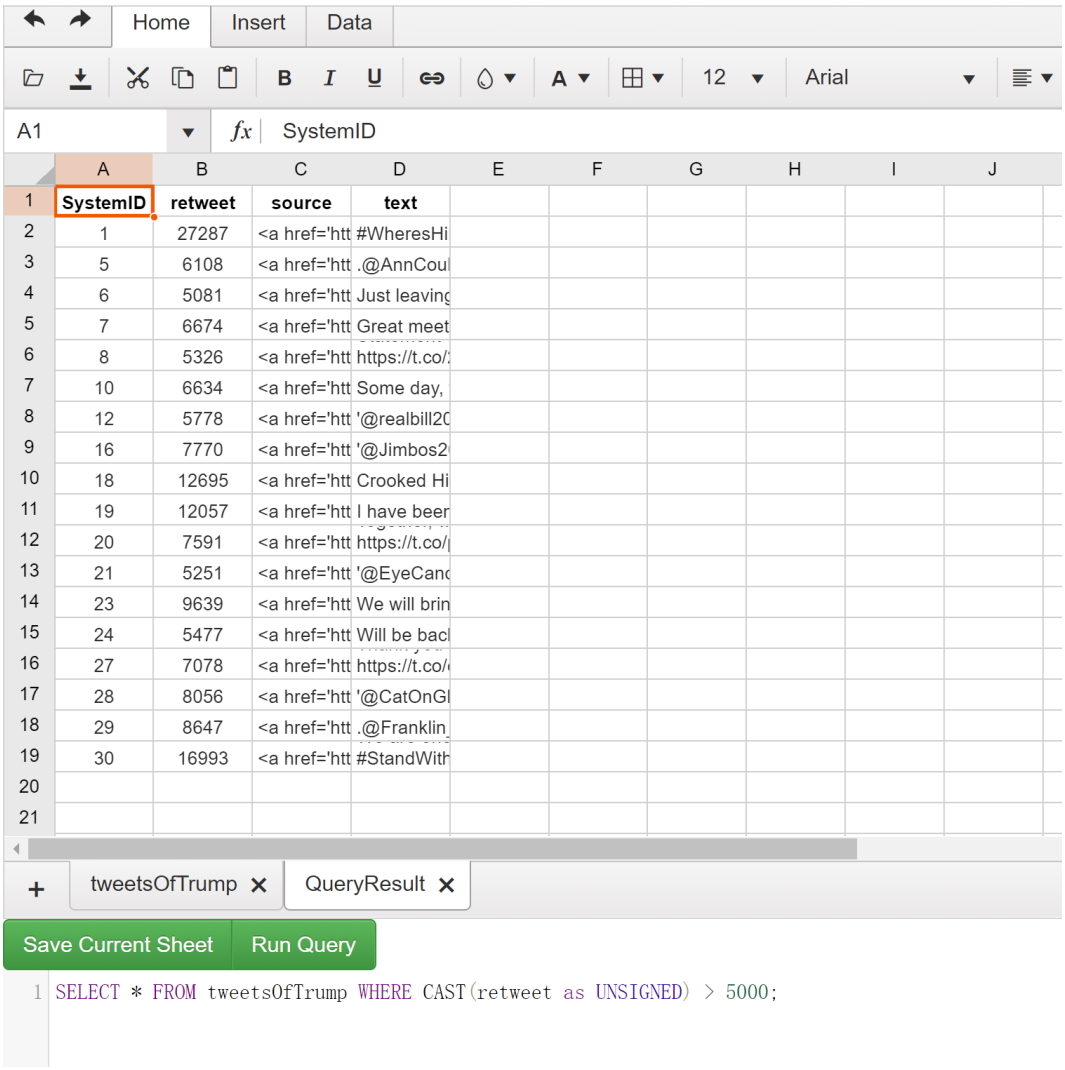

Figure 3.12: Example of MySQL query

4. FUTURE WORK

4.1 Data Parsing Improvement

Currently when parsing data, the Lion-Backedn server will iterate the whole JSON data layer by layer to create an object, get the ID and pass to the next layer as the parent's ID. Such an algorithm completes the traversal serially. Thus, when the size of the data is large, the performance could be affected. In this case, optimizing the parsing algorithm to parallelize the traversal is one of the possible improvements.

4.2 MySQL Namespace Improvement

As mentioned in Chapter 3, data in the spreadsheet are stored in MySQL with table name $derivedTableName$ instead of $originalTableName$. In order to support use of $originalTableName$ in queries, the query finally being executed is

*WITH (SELECT * FROM* $derivedTableName1$*) AS* $originalTableName1$*,*
*(SELECT * FROM* $derivedTableName2$*) AS* $originalTableName2$*,*
*(SELECT * FROM* $derivedTableName3$*) AS* $originalTableName3$*,*
...
$userQuery$

where $userQuery$ is the actual query entered by the user. Obviously, this query will decrease efficiency when there are many tables and a large amount of data. Also, the common table expression syntax (i.e. *WITH...AS...*) is only supported by MySQL 8.x or above [12]. Therefore, figuring out another approach to implement the namespace functionality can improve the performance and compatibility of the system.

5. CONCLUSION

Lion-Backend aims to solve the problem of unstructured social media data storage and processing. In Lion-Backend, the problems are solved step by step. The unstructured data understanding (or parsing) problem is handled by an unstructured JSON data parser, and the Neo4j database works for the storing problem. For the processing part, visual query builder helps users to query their data easily and efficiently. Multiple tools including visualization and text analysis are provided. For those users who have a stronger background in programming, customized formulas are provided as an advanced option. In the end, based on the similarity between the spreadsheet and MySQL table, Lion-Backend fully makes use of MySQL as a platform to perform more complex processing operations and store the processing result permanently. Therefore, it is shown that with Neo4j and MySQL integrated together, Lion-Backend is a solution for unstructured JSON data storage and processing.

REFERENCES

[1] A. Shahjahan and K. U. Chisty, "Social media research and its effect on our society." *World Academy of Science, Engineering and Technology International Journal of Information and Communication Engineering*, vol. 8, no. 6, 2014.

[2] IBM Corporation, "What is a database management system?" 1990-2010. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/zosbasics/com.ibm.zos.zmiddbmg/zmiddle_46.htm

[3] B. Dickinson, "So what the heck is the 'social graph' Facebook keeps talking about?" Mar 2012. [Online]. Available: https://www.businessinsider.com/explainer-what-exactly-is-the-social-graph-2012-3

[4] E. Hart, P. Barmby, D. Lebauer, F. Michonneau, S. Mount, P. Mulrooney, T. Poisot, K. H. Woo, N. Zimmerman, and J. W. Hollister, "Ten simple rules for digital data storage," *PLOS Computational Biology*, Oct 2016. [Online]. Available: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005097

[5] "Neo4j graph platform the leader in graph databases." [Online]. Available: https://neo4j.com/

[6] J. Wolverton, "Genson." [Online]. Available: https://pypi.org/project/genson/

[7] "Popoto.js." [Online]. Available: http://popotojs.com/

[8] "The py2neo v4 handbook." [Online]. Available: https://py2neo.org/v4/

[9] J. M. Wagner and J. Keisler, "Enhance your own research productivity using spreadsheets," *Models, Methods, and Applications for Innovative Decision Making*, p. 148162, 2006. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/educ.1063.0028

[10] "Text analysis API — natural language API." [Online]. Available: https://aylien.com/text-api/

[11] K. W. Broman and K. H. Woo, "Data organization in spreadsheets," *The American Statistician*, vol. 72, no. 1, p. 210, Sep 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1375989

[12] "MySQL 8.0 reference manual: 13.2.13 with syntax (common table expressions)." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/with.html